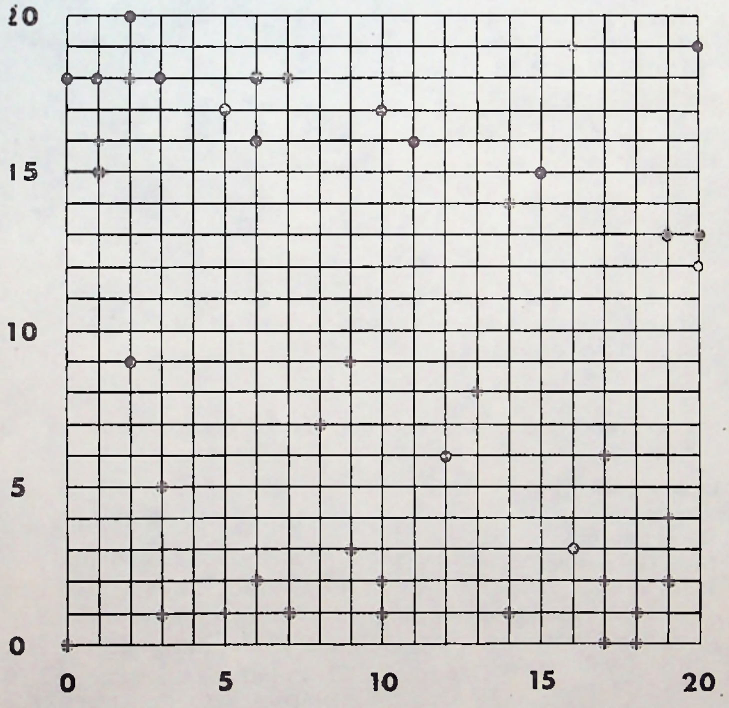
Popular Computing

Volume 8
Number 8
August 1980

89

The world's only magazine devoted to the art of computing.



NOSQUARE

Sylvia Leonard, UCLA, actually programmed the TIME Game (issue number 87), even to the extent of simulating the crooked dice. The result of playing the game 2719 times was this distribution:

5	3	11	294
6	2	12	433
7	195	13	183
8	619	14	11
9	577	15	59
10	342	16	1

that is, the game was ended after 5 moves just 3 times; after 16 moves, just once. No game took more than 16 moves to complete. □

Publisher: Audrey Gruenberger
Editor: Fred Gruenberger
Associate Editors: David Babcock
Irwin Greenwald
Patrick Hall
Contributing Editors: Richard Andree
William C. McGee
Thomas R. Parkin
Edward Ryan
Art Director: John G. Scott
Business Manager: Ben Moore

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each. Copyright 1980 by POPULAR COMPUTING.

Case Study 2

NOSQUARE

The lattice shown on the cover has 21 x 21 lattice points. 43 of these 441 points have been selected so that no four of them form a square.

We will proceed to examine how such a pattern can be obtained.

First, let's back off to consider some related facts about the situation. On a lattice of 441 points, the total number of ways that 4 points can be selected is:

$$_{441}C_4 = \frac{441 \cdot 440 \cdot 439 \cdot 438}{4 \cdot 3 \cdot 2 \cdot 1}$$

$$= 1{,}554{,}599{,}970$$

or, a billion and a half sets of 4 points. Most of those sets do NOT form a square, of course.

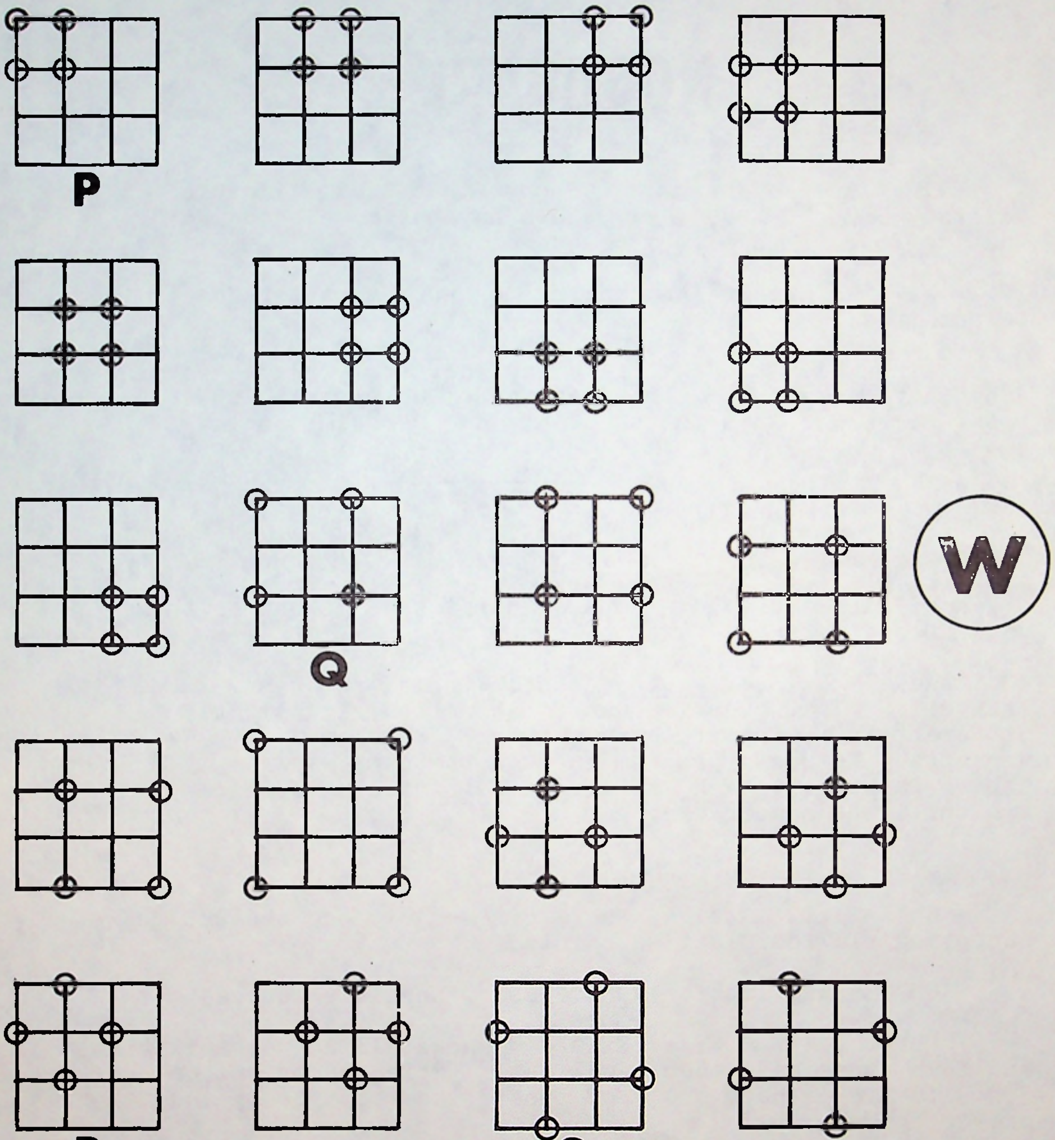
Well, how many squares can be formed on such a lattice? That question was first raised, in slightly different form, in our issue number 7, Problem 17. We can attack this sub-problem in much the same way as Thomas Parkin attacked the Circle Partitioning problem (in our issue number 84).

On a 2 x 2 lattice, there can be just one square, occupying all four lattice points.

On a 3 x 3 lattice, there are 6 possible squares.

For the 4 x 4 case, Figure W shows all 20 possibilities, and four types have been labelled. Going back, on the 3 x 3 lattice, there are 4 of type P, one of type Q, and one of type R.

Similar analysis shows 50 possible squares on a 5 x 5 lattice, and 105 on a 6 x 6 lattice. We have sufficient data to form the difference table shown in Figure V.


P
Q
W
R
S

V

CASE	No.	1 st diff's	2 nd diff's	3 rd diff's	4 th diff's
2	1				
		5			
3	6		9		
		14		7	
4	20		16		2
		30		9	
5	50		25		
		55			
6	105				

There is not enough data, however, to establish anything (we would like to have a column of differences constant). But the second differences now look suspiciously like successive squares and, if so, the fourth differences would be constant. We would have then established, at least crudely, that the function we are seeking is a polynomial and is of 4th degree.

If we throw in, now, the "one" case (on a 1 x 1 lattice, the number of squares is zero), then the second difference column begins with 4:

1	0				
		1			
2	1		4		
		5		5	
3	6		9		2
		14		7	
4	20		16		2
		30		9	
5	50		25		
		55			
6	105				

and our conjecture is reinforced. If we feel especially cautious, we could count the possible squres on a 7 x 7 lattice (there are 196 of them) and increase our confidence. In any event, we can then apply the Newton-Gregory interpolation formula, as Mr. Parkin did, or we can use the method of undetermined coefficients. Assuming that the desired formula is of the form:

$$y = ax^4 + bx^3 + cx^2 + dx + e,$$

we can substitute five known values, to determine the five unknown coefficients. The first of these can be (1,0), which makes e = 0 immediately. Then we have:

$$\begin{cases} 1 = 16a + 8b + 4c + 2d \\ 6 = 81a + 27b + 9c + 3d \\ 20 = 256a + 64b + 16c + 4d \\ 50 = 625a + 125b + 25c + 5d \end{cases}$$

and this system solves to produce:

$$a = 1/12 \qquad b = 0 \qquad c = -1/12 \qquad d = 0$$

so that we have the formula:

$$y = \frac{x^4 - x^2}{12}$$

From this formula we can find that, on a 21 x 21 lattice, there are 16170 sets of four points that form a square.

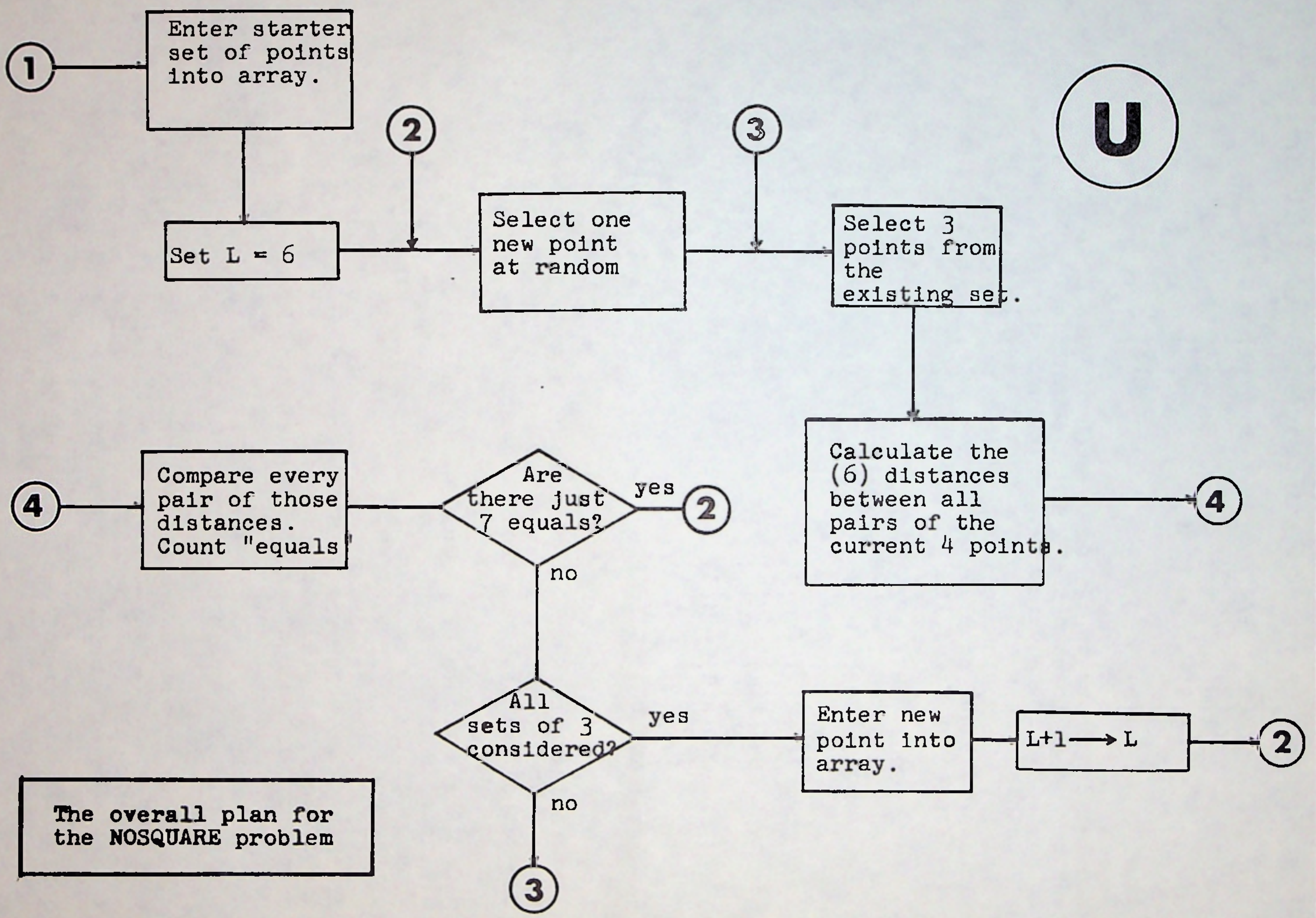
We will describe a straightforward attack on the problem, as shown in Flowchart U.

First, we set up 6 points that form a starter set; that is, a set of points for which we know (by observation) that no four of them form a square. These points are entered into two arrays, X and Y (keeping the X and Y coordinates separate, for convenience).

The selection of a random point, on a 21 x 21 lattice, may be effected by instructions of the form:

```
X = INT(21 * RND(1) )
Y = INT(21 * RND(1) )
```

suitably modified to conform to the rules of the BASIC you use.


1
Enter starter set of points into array.
Set L = 6
2
Select one new point at random
3
Select 3 points from the existing set.
U
Calculate the (6) distances between all pairs of the current 4 points.
4
4
Compare every pair of those distances. Count "equals"
Are there just 7 equals?
yes
2
no
All sets of 3 considered?
yes
Enter new point into array.
L+1 ⟶ L
2
no
3

The overall plan for the NOSQUARE problem

The starter set of 6 points might be the following:

```
X(1) = 17            Y(1) = 0
X(2) = 18            Y(2) = 0
X(3) = 19            Y(3) = 0
X(4) = 20            Y(4) = 17
X(5) = 20            Y(5) = 18
X(6) = 20            Y(6) = 19
```

and two other arrays, A and B, each of just 4 elements, can be set up to hold the X and Y coordinates of 4 points to be tested for squareness. The process of selecting all possible sets of 3 points from those already known (that is, the action called for at Reference 3 of the flowchart) to be coupled with a new point under test, can be effected by steps like the following:

```
2000 FØR I = 1 TØ (L-2)
2010 FØR J = (I+1) TØ (L-1)
2020 FØR K = (J+1) TØ L

2030 A(3) = X(K)
2040 B(3) = Y(K)
2050 A(2) = X(J)
2060 B(2) = Y(J)
2070 A(1) = X(I)
2080 B(1) = Y(I)
```

with A(4) and B(4) being supplied with the newly selected random point.

All of the above, then, supplies 4 points which are to be tested to determine whether or not they form a square. One method of attack is the following.

For 4 points, P, Q, R, and S, calculate the squared distances PQ, PR, PS, QR, and QS. Compare each of these squared distances to each of the others (making 15 such comparisons). If P, Q, R, and S form a square, then exactly 7 of those comparisons will come out "equal." If that is so, then the point under test may be rejected immediately. Otherwise, we test to see if all sets of 3 points (from our stock of known points) have been now used up. If all sets of 3 have been considered, then the new point is added to the X and Y arrays, and the value of L (the number of points in the pool) is increased by <u>one</u>.

This procedure becomes ever more costly in time as each new point is acquired. From the statistics of the situation (only 16170 possible squares out of a billion and a half sets of 4 points), one might expect a flood of rejected points, but actually they are rare, at least during the sieving of the early points. A rough approximation to the time of calculation yields:

$$y = 6x^2 - 11x + 20$$

where x is the number of the point and y is the time in seconds. Thus, the first point will be found in about 15 seconds; the tenth point will take over 8 minutes; and the 25th point will take nearly an hour.

It is not known how many points there could be on such a lattice under these conditions.

* * * * *

The NOSQUARE problem lends itself ideally to being a cooperative project for an advanced class in computing. Pages labelled (*) show handouts that were used in such a project for a class in Numerical Methods. During the course of this project, a diary was maintained, from which the following was excerpted.

* * * * * * * * * *

(February 6) The problem was presented to the class and the sheets labelled (*) were handed out. The initial howls were disposed of:

> "Can some of us work in SIMULA?" No way.
>
> "Is it all right to have several people work together?" All you like, but each student is to write an independent program and be responsible for it.

(February 13) The breakdown of the overall problem into six sub-parts was discussed. The seventh subroutine, SUB7 (a random number generator) was written by the instructor and copies of the assembly listing of that subroutine were distributed as a model of what is to be done. A set of standards for this project was introduced and discussed.

The logic of testing for squareness was gone over, as well as the mathematics of the number of possible squares, and the expectation of how many points might actually be produced when and if the final program runs.

New examples of independent thinking pop up:

"In subroutine 4, could we call a variable DSQ rather than SUB498?" Well, no; as soon as a little of such creativity is tolerated on a large job, no one will ever be able to read anyone else's code. Name it according to the project's standards and explain its function to your heart's content in REMARKS.

"May we use the special linking instruction of our assembler?" How could this be? The one who writes a subroutine and the one who links to it are not the same person. There has to be some agreement on such things. The only intelligent way is to establish some sort of standard (which was done) and then insist that everyone follow it.

(February 25) A lottery was held to assign the six routines to the members of the class; they have had time to look them over and form preferences. There were at that time 22 students in the class; they were given numbers from 1 to 22. Using a programmable pocket calculator, random integers in the range from 1 to 22 were generated, to select a random ordering of the students. Each student could then select which routine he wished to work on, subject to the constraint that no more than four students could be assigned to any one routine. Thus, even the last student selected in the lottery still had a choice of three routines to pick from. Incidentally, since the random number generator used in this lottery produces duplicate numbers, as it should, there is a vivid demonstration of randomness, given at a time when it is meaningful.

The essence of the project is cooperation between the writers of the various routines--a tiny simulation of a part of the real world of computing that is neglected in most curricula. Thus, if subroutine 3 signals when all sets of 3 points have been utilized, the other people must know where that signal is to be found. Therefore, all those working on SUB3 should agree on, say, SUB395 as the word to contain that signal, with the understanding that if SUB395 is zero it means "done" and if SUB395 is nonzero it means "not done." It takes about a week for the thought to sink in that there must be such agreement, since any one of the decks produced for SUB3 may be involved with any of the other routines. So after the SUB3 people agree on such things, then whatever they decide must be communicated to everyone else.

All of this leads to the information sheets that were mentioned earlier. At the peak point in such a project, new information sheets will be coming out every class day. The instructor took on the role of project leader for much of this information exchange, to forestall conflicting suggestions (or outright foolish ones). For example, SUB2 is to check that a randomly selected point has not been previously selected. One way to monitor this action is to create a bit map of 441 bits and store a one in the proper bit position of any selected point. This is quite logical, but somewhat inefficient (there are lots of ways to do it). It is also most likely going to be a mess to code, debug, and test, and so the suggestion was quietly killed, after much class discussion of the pros and cons.

(March 10) Clever schemes to save a microsecond are emerging, long before any routines are known to work, and in particular before it is known if they work with other routines. The aphorism:

FIRST MAKE IT WORK, THEN MAKE IT PRETTY

suddenly makes good sense. Questions of responsibility are being raised; namely, "who is supposed to clear out these arrays, and when?" Questions are also coming up that indicate that these (supposedly advanced) students are not too well grounded in the fundamentals of their major subject. Questions like these:

>> How many index registers does our machine have?

>> Is it possible to index down?

>> How many words of storage are available to us for this program?

>> Who will furnish the job control cards for our running decks?

--make one wonder just what they have been learning prior to this class.

(March 24) On this day, each student was to hand in an assembly listing, showing that his routine was debugged, documented properly in its REMARKS, and tested to insure that it does what it is supposed to do. They were also to submit a deck, with all of its job control cards removed, as well as any instructions that had been used solely to test the routine.

This is, of course, chaos day. Some students still have no notion of what program testing means. Most of them had no idea as to which cards to delete from their deck. Few of the decks were identified as to their function or their author, and nearly every deck contained clear cut violations of the agreed-upon standards. This seemed like a good time to offer a short lecture about large-scale programming projects, like the one to get a man to the moon AND BACK.

(March 26) The listings were returned, graded, with sassy comments in red:

- Where are the REMARKS for these instructions?
- There is no SUB8, so don't link to it.
- What do two "LOAD ACCUMULATOR" instructions in a row *do*?
- This routine printed four numbers--should we guess what they are? Do you know?
- Since you used your own quaint escape mechanism, no one else can possibly link to your subroutine.

--and so on. Color their faces red today.

(April 9) From the collection of decks submitted, three run decks were made up and sent off to the computer. The resulting collection of error messages was presented to the class, together with the assembly listings. All sorts of interesting anomolies are revealed. The three run decks include 17 of the class's decks.

No two of the routines are identified in the same place, although the proper format for such identification was defined in one of the information sheets.

Our assembler calls for the following card layout:

Location symbol or data label	Columns 1-8, left justified
Operation code	Columns 10-18, left justified
Address or value	Columns 20-72, left justified.

One student has discovered that the assembler will accept addresses starting in column 19; his cards sort of stand out from the mob. Another has found that the "left justified" requirement for location symbols is not rigid, so his symbols wander in and out.

Here's a student who spent all his time working on alphabetic headings for his routine's output. He never had any output (he never got that far); moreover, his misguided efforts require serious changes in the JCL for the entire deck.

Correction cards have been arriving at the instructor's mailbox and via the department secretary--unlabelled as to which deck, which routine, or which student.

The three running decks are still loaded, after numerous submissions, with JCL cards left over from individual test runs. Most of these tend to truncate the running of the deck.

Standard practice calls for variables and constants of a subroutine to be at the end of the subroutine. The type of permissive thinking that puts them in front of a subroutine (or, good grief!, in the middle) just has to go.

All in all, one big mess to be worked on.

(April 16) An interesting new idea: suppose that one of our running decks eventually gets to work and produce real results--shouldn't we plan on a RESTART procedure to allow successive runs to capitalize on previous results?

We now had four working decks. The fourth one was made by reproducing one of the others (the one that was now giving the least trouble) and swapping in the last remaining subroutines. Every student deck is now in one of the running decks.

Our assembler (which is one of the world's worst) does give fairly clear indications of trouble in most cases. However, when a program manages to jump to address 0000 (usually this occurs because a supplied address in a subroutine escape has not been properly supplied), it is difficult to locate the error. Weird debugging techniques are being invented, including some elaborate tracing schemes. However, on the whole the class is now highly receptive to discussions of good coding practices.

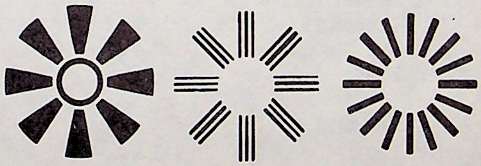
(April 24) How time flies when you're having fun. We are now getting four fat printouts a day; one of the four working decks is now up to try number twelve. Deck 3 is producing output, but patently false output (the point 0,0 four times).

"Corrections" are being made to the decks that trigger batches of new error messages. The core dump that came back with one printout contained a mysterious table of 4200 entries that no one could identify.

One listing indicated that the default option on printing (500 lines) had been exceeded. The number of lines was changed to 750 and the program on its next run bombed off the machine at address 0000.

(May 7) The four working decks have racked up collectively 38 runs. One of the decks suddenly produced real results; namely, exactly four points, followed by a jump to 0000.

It is time to quit. Nothing further of interest would be learned by continuing, and there is certainly ample material with which to assign grades. When it is announced that we have come to quitting time, suddenly there is great interest expressed in going on. One is reminded of crocodile tears.



Cooperative Project **NOSQUARE** Computer Science

Spring 1980

The Problem: On a 21 x 21 lattice, we wish to find a collection of points, most of them randomly selected, such that no set of four of the points forms a square.

The overall task has been broken into six parts, and each student will be responsible for one of the parts. Each student is to write his own program, which will be graded. The following is to be done:

1. Analyze your sub-problem and draw flowcharts as needed.

2. Write a program in assembly language, following standard practices.

3. Within the program, include sufficient REMARKS to identify which sub-program it is, what it purports to do, and the name of its author.

4. Label carefully those parts, or features, or words in the sub-program that may have to be referred to by others.

5. Each subroutine is to be CLOSED; that is, it will be linked to from the MAIN program. It should contain all of its own data. Any variables, constants, or tables that are common to several routines, will be defined in the MAIN program.

6. Arrange to TEST your program. This involves devising dummy data to use, and perhaps faking various operations of other routines. Aim to demonstrate, to your satisfaction and mine, that your program:

   (a) Does what it is supposed to do.

   (b) Does not do things it is not supposed to do.

   (c) Will continue to work properly when the data changes.

While you are testing your program, you may be operating with many extra instructions that must be deleted when we put together complete run decks. Be sure to identify such instructions with copious REMARKS.

The separate routines are to be named MAIN, SUB1, SUB2, SUB3, SUB4, SUB5, and SUB6. The name of a routine is also the symbolic location of its first instruction and is the only entry point to the subroutine.

All reference names within a subroutine (or MAIN) are to be formed out of the name of the routine; for example, SUB52, SUB37, and so on. Any data words in a routine are to be named as SUB299, SUB697, and so on. The exit point from any subroutine shall be Reference 9 and have a label such as SUB49.

The linkage to a subroutine will be:

```
        LDA     *
        UJP     SUB3
```

and within a subroutine the escape mechanism will be:

```
SUB3    INA     2         (Identifying
        SWA     SUB39     REMARKS go here)
        ...     .....

(body of the subroutine here)

SUB39   UJP     0000
```

Information sheets will be distributed from time to time, to keep everyone informed of what is going on; therefore, you keep the instructor informed of what you are doing.

SUB2 needs a random number generator. This will be SUB7, written by the instructor.

The goals of the project are these:

- Proper organization of cooperative computing work.
- Conformity to established standards.
- Familiarity and practice with assembly language.
- Recognition of the need for communication among co-workers in computing.

Project NOSQUARE *

MAIN	The MAIN (controlling) program; does all the initialization; calls all other subroutines as needed; contains all data common to other subroutines; sets up starting pool of known points in arrays X and Y.
2	SUBROUTINE to select a point at random on the 21 x 21 lattice. Check that the point selected is not already in the pool of known points and, if so, select another.
3	SUBROUTINE to select systematically a set of 3 points from the current pool of points on demand. Provide a signal when all sets of 3 have been used.
4	SUBROUTINE to form the squared distances between each pair of the current 4 points (namely, the 3 points furnished by Subroutine 3 and the point furnished by Subroutine 2). There are 6 of these squared distances.
5	SUBROUTINE to compare every pair among the 6 distances furnished by Subroutine 4 (there are 15 comparisons). Furnish a count of the number of comparisons that are "equal."
6	SUBROUTINE to accept or reject the point being considered. If accepted, print it, suitably identified, and add it to the pool. If rejected, print it, together with the three other points that make a square.

Problem Solution

Problem 155 (Friedberg's Sequence) in issue 46 was the following:

> For each positive integer, N, form S, the smallest factor of $4N^2+1$. What is the sum of the S values for the first 1000 values of N?

The problem is taken from Richard Friedberg's An Adventurer's Guide to Number Theory (McGraw-Hill, 1968). The following is demonstrated there: S can never be 2 or 3, and, for values of N of the form 5K+1 or 5K-1, S is 5.

Ralph Montgomery, of St. John's Community College, produced this result: 254 297 228. His flowchart is given, slightly paraphrased, as Figure D. Montgomery used the notation of S for the sum, and F for the values that Friedberg called S in the original problem.

As can be seen from the flowchart, Montgomery chose to utilize the fact that S is 5 when N is of the form 5K+1 or 5K-1. His variables K1 and K2 keep a running record of numbers of that form. For those numbers, it is not necessary to test (at Reference 3), which realizes a saving. Since K1 is always less than K2, the updating of both of them can be done when N = K2.

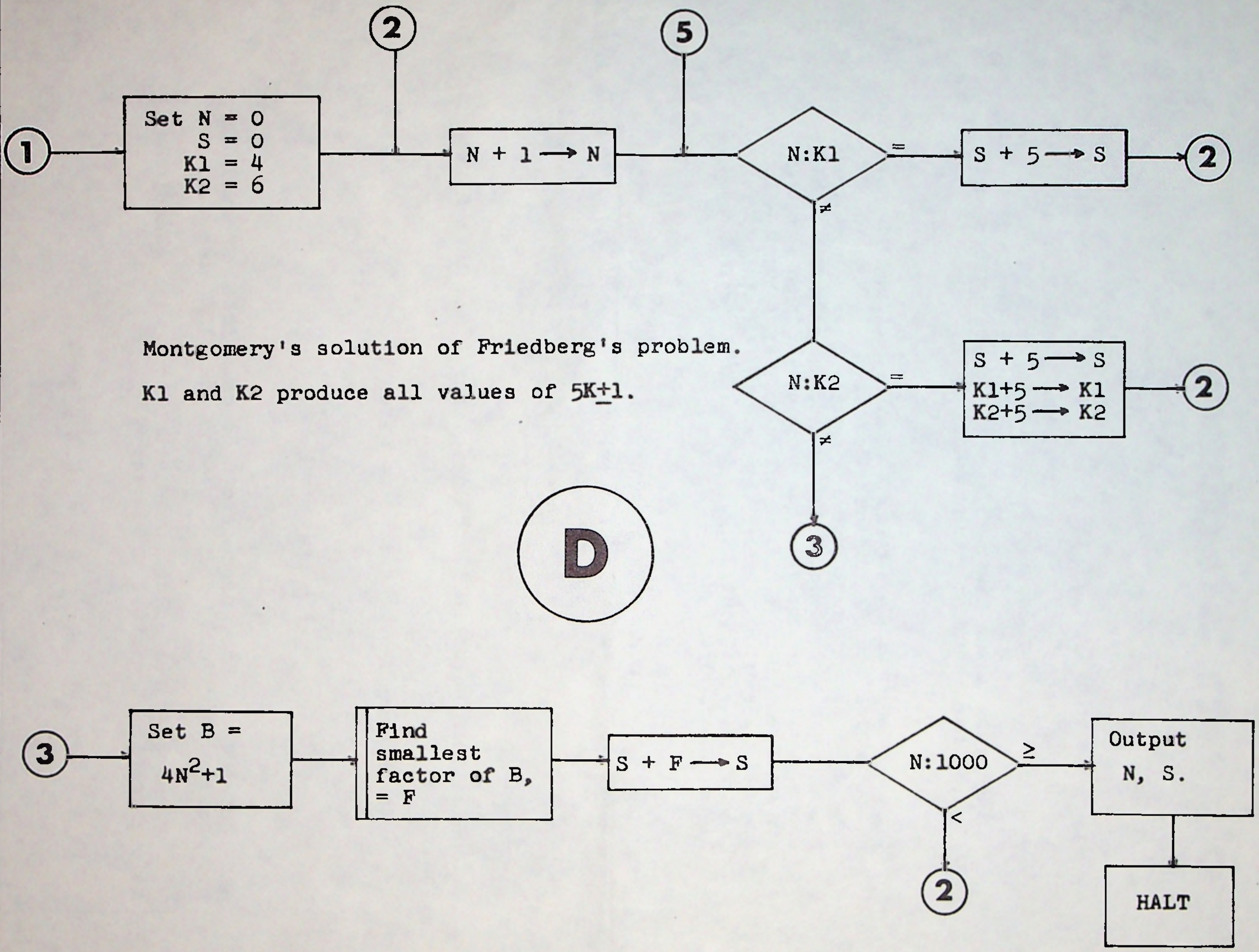
The question that Mr. Montgomery raises is, is it worth it to incorporate such shortcuts into a program? His solution was obtained in BASIC, so the logic at Reference 5 costs two IF statements, plus the updating statements for K1 and K2, and all of this saves just one division in the factoring subroutine, which is written to begin searching for factors with 5 (Friedberg states that 2 and 3 cannot be factors).

Such questions of efficiency pop up all over in computer programming. There are always three things to be weighed:

1. The work involved in programming the suggested shortcut.

2. The possible saving, usually of CPU time.

3. The complexity factor--how will it affect the maintainability of the program?



Montgomery's solution of Friedberg's problem.

K1 and K2 produce all values of $5K\underline{+}1$.

Problem Solution

John W. Wrench, Jr. has furnished a solution to Problem 53-C (in our issue number 16).

The problem was taken from the book Computers and Society (Richard Hamming, McGraw-Hill 1972):

> Suppose we start with a circle of radius one, and we draw around it an equilateral triangle (circumscribed triangle), and around the triangle we draw another circle. Around this circle we draw a square, and around the square we draw another circle. Then we draw a regular five-sided polygon and a circle, then a regular six-sided polygon and a circle, and so on, continuing indefinitely. How will the radii of the circles behave? Will they approach infinity (get arbitrarily large), or will they remain bounded by some finite size?

Using the notation R_3 for the radius of the circle that circumscribes the equilateral triangle, it is easy to show that:

$$R_n = \frac{R_{n-1}}{\cos(\pi/n)} \qquad (A)$$

...and at this point in issue 16 we wrote "and that R_n tends toward the value 8.6996 as n increases."

Mr. Wrench shows that R_n is given by:

$$\prod_{n=3}^{\infty} \left(\cos \frac{\pi}{n}\right)^{-1}$$

and that the infinite product can be calculated analytically. His result is: 8.70003 66252 08194 50322 24117 144...

A brute-force calculation, iterating with equation (A) to 9 significant digits, produces:

$$R_{30400} = 4.34933284$$